

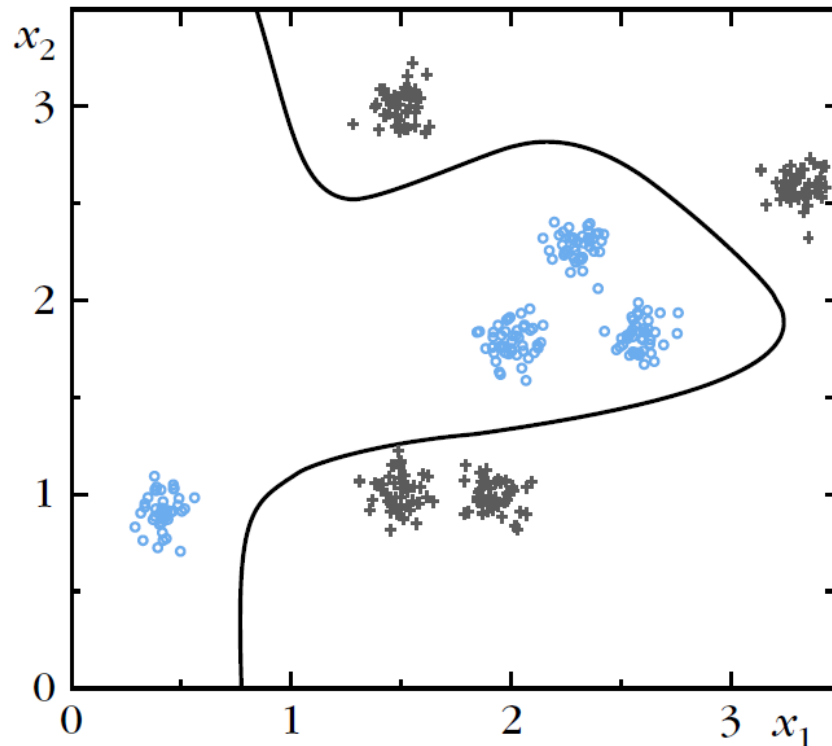
KECE470 Pattern Recognition

Chapter 4. Nonlinear Classifiers

Chang-Su Kim

Nonlinear Classifiers

We now deal with problems that are not linearly separable and for which the design of a linear classifier does not lead to satisfactory performance



ONE! TWO! THREE!

One-Layer Perceptron

- XOR problem is not linearly separable

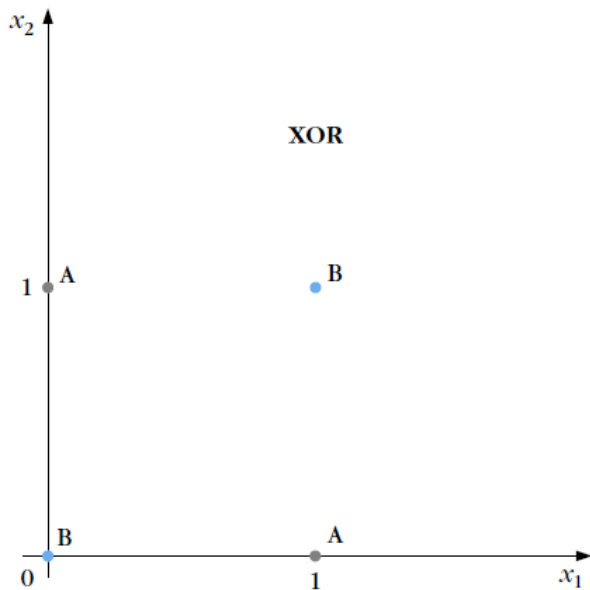


Table 4.1 Truth Table for the XOR Problem

x_1	x_2	XOR	Class
0	0	0	B
0	1	1	A
1	0	1	A
1	1	0	B

One-Layer Perceptron

- AND and OR problems are linearly separable

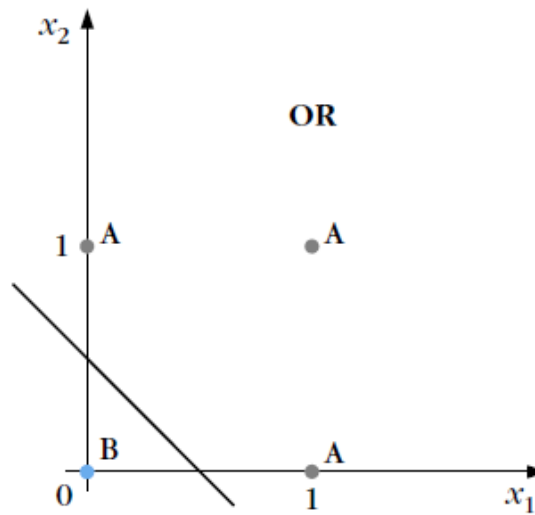
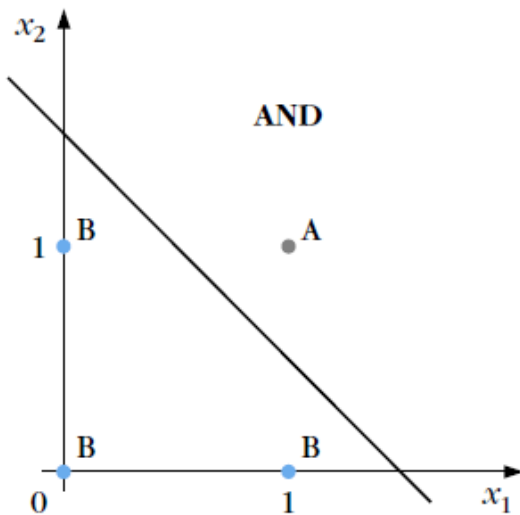
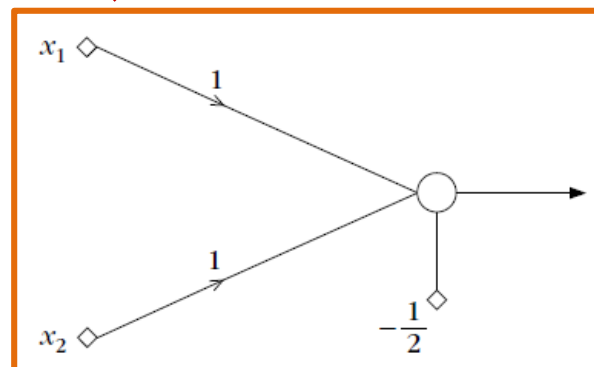


Table 4.2 Truth Table for AND and OR Problems

x_1	x_2	AND	Class	OR	Class
0	0	0	B	0	B
0	1	0	B	1	A
1	0	0	B	1	A
1	1	1	A	1	A

↓
1-layer perceptron
implementation



Two-Layer Perceptron

- XOR problem: solve it in two successive phases
 - 1st phase (or layer) uses two lines

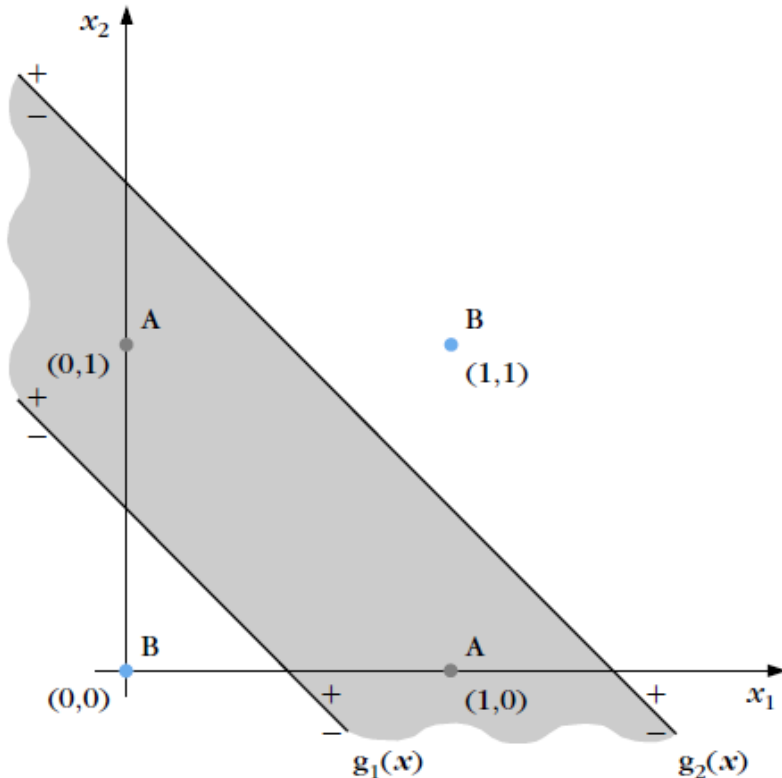


Table 4.3 Truth Table for the Two Computation Phases of the XOR Problem

		1st Phase		2nd Phase
x_1	x_2	y_1	y_2	
0	0	0 (-)	0 (-)	B (0)
0	1	1 (+)	0 (-)	A (1)
1	0	1 (+)	0 (-)	A (1)
1	1	1 (+)	1 (+)	B (0)

Two-Layer Perceptron

- XOR problem: solve it in two successive phases
 - 2nd phase

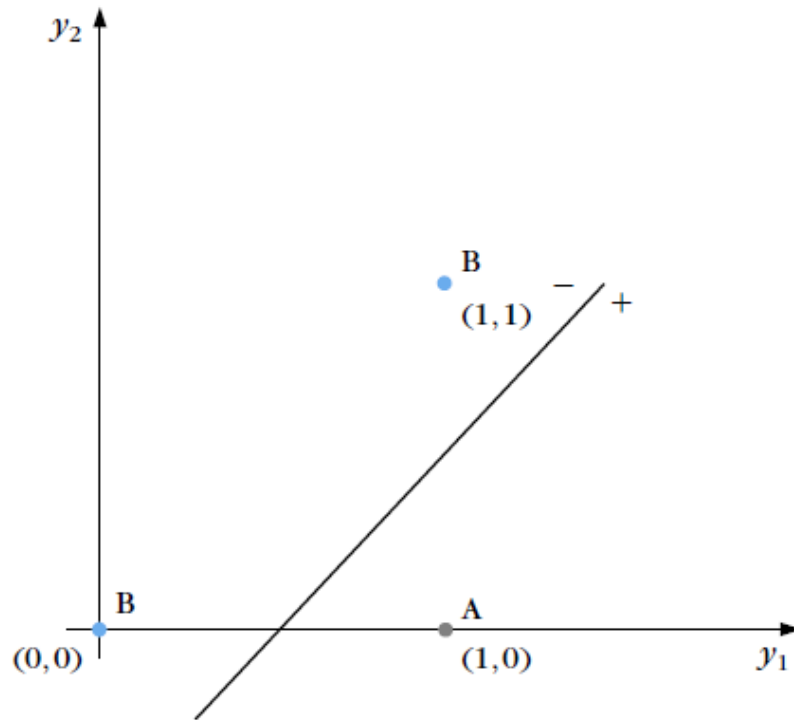


Table 4.3 Truth Table for the Two Computation Phases of the XOR Problem

		1st Phase		2nd Phase
x_1	x_2	y_1	y_2	
0	0	0 (-)	0 (-)	B (0)
0	1	1 (+)	0 (-)	A (1)
1	0	1 (+)	0 (-)	A (1)
1	1	1 (+)	1 (+)	B (0)

Two-Layer Perceptron

- XOR problem: solve it in two successive phases
 - 2-layer perceptron (or 2-layer feedforward neural network)

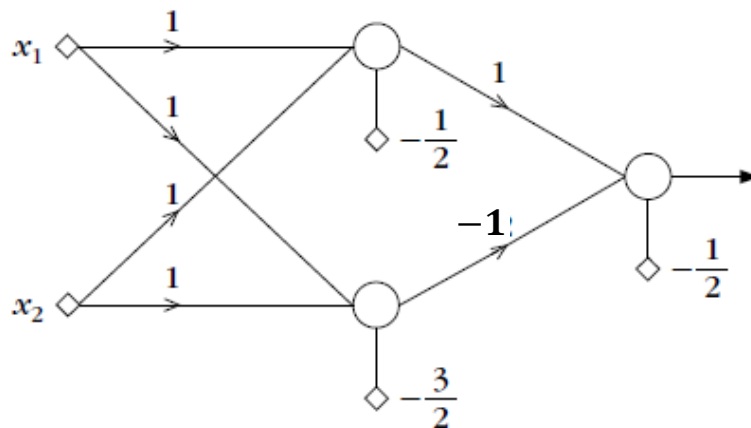


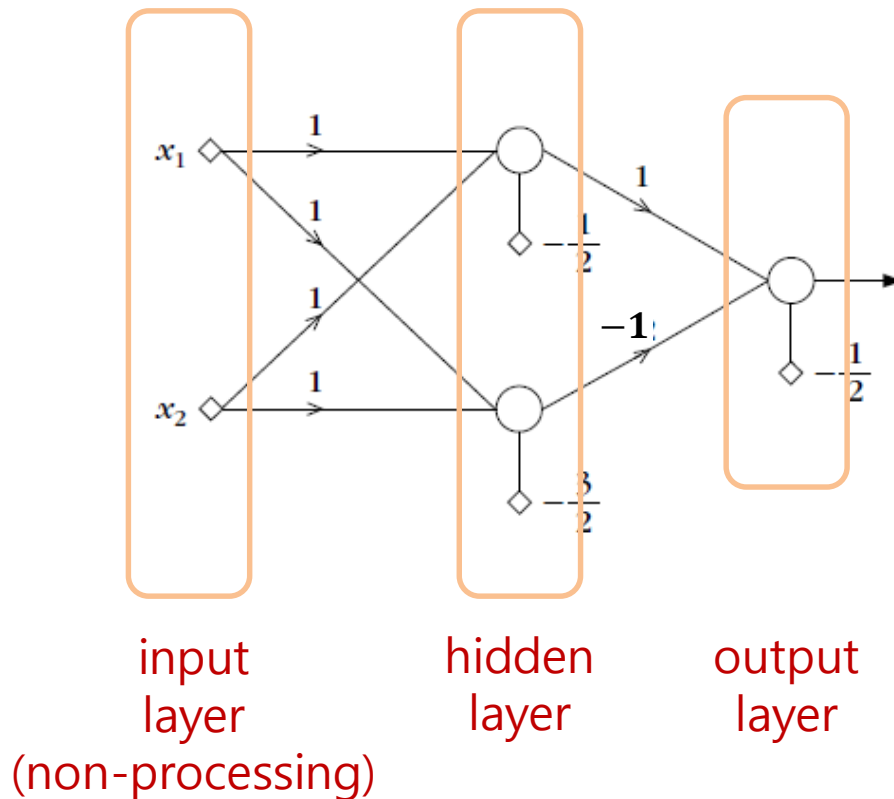
Table 4.3 Truth Table for the Two Computation Phases of the XOR Problem

		1st Phase		2nd Phase
x_1	x_2	y_1	y_2	
0	0	0 (-)	0 (-)	B (0)
0	1	1 (+)	0 (-)	A (1)
1	0	1 (+)	0 (-)	A (1)
1	1	1 (+)	1 (+)	B (0)

- $g_1(\mathbf{x}) = x_1 + x_2 - \frac{1}{2} = 0$
- $g_2(\mathbf{x}) = x_1 + x_2 - \frac{3}{2} = 0$
- $g(\mathbf{y}) = y_1 - y_2 - \frac{1}{2} = 0$

Two-Layer Perceptron

- Terminology
 - 2-layer perceptron (or 2-layer feedforward neural network)

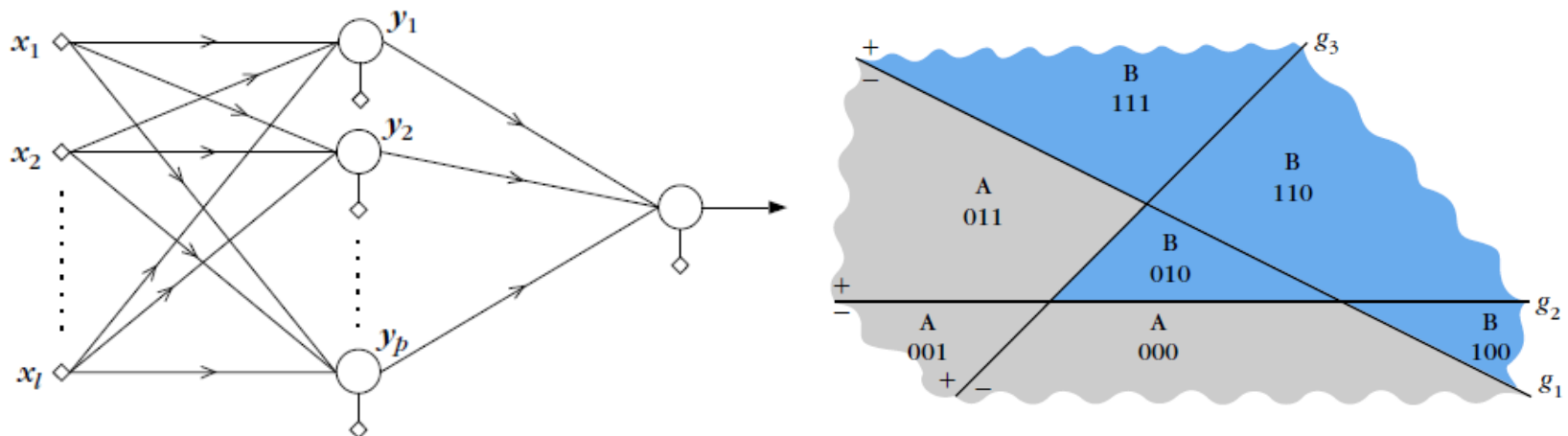


Two-Layer Perceptron

- Classification capabilities of two-layer perceptron
 - 1st layer maps input to vertices of the unit hypercube

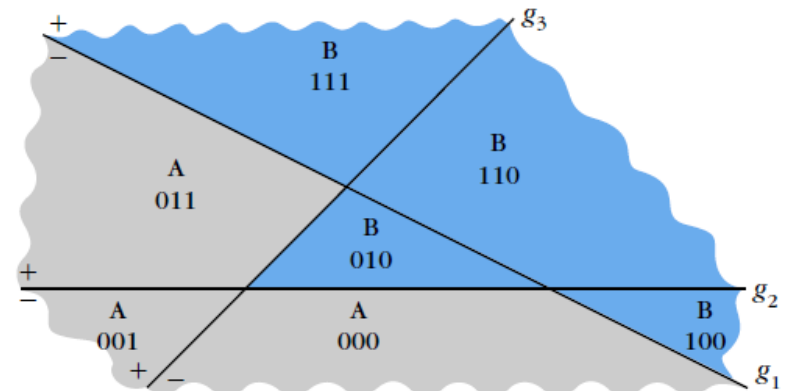
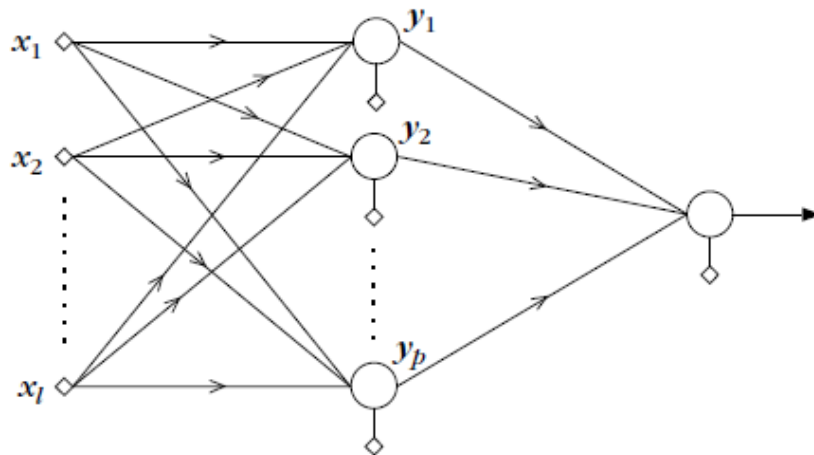
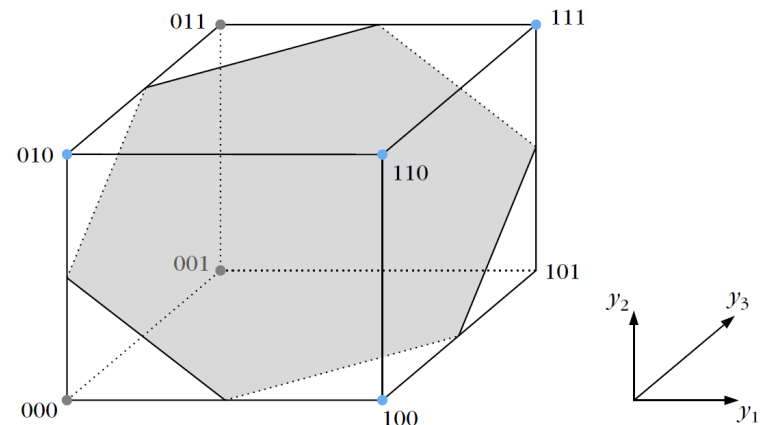
$$H_p = \{[y_1, \dots, y_p]^T \in \mathbb{R}^p: y_i \in [0, 1] \text{ for } 1 \leq i \leq p\}$$

- An output of 1st layer corresponds to a polyhedron



Two-Layer Perceptron

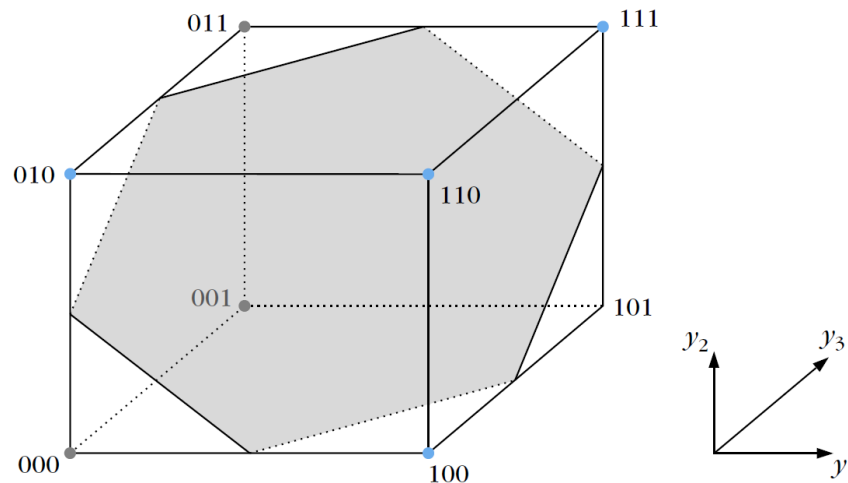
- Classification capabilities of two-layer perceptron
 - 2nd layer detects a union of selected polyhedra



Two-Layer Perceptron

- Classification capabilities of two-layer perceptron

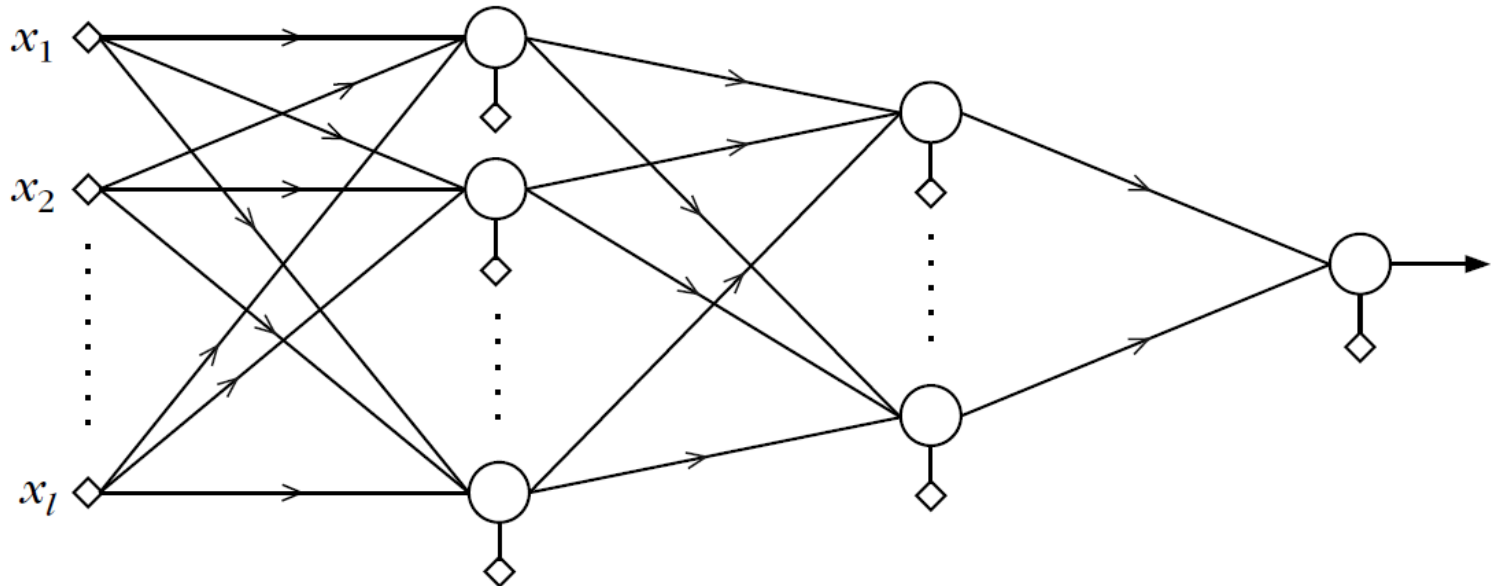
Two-layer perceptron can detect a class, which consists of a union of polyhedral regions, but not any union of such regions



Three-Layer Perceptron

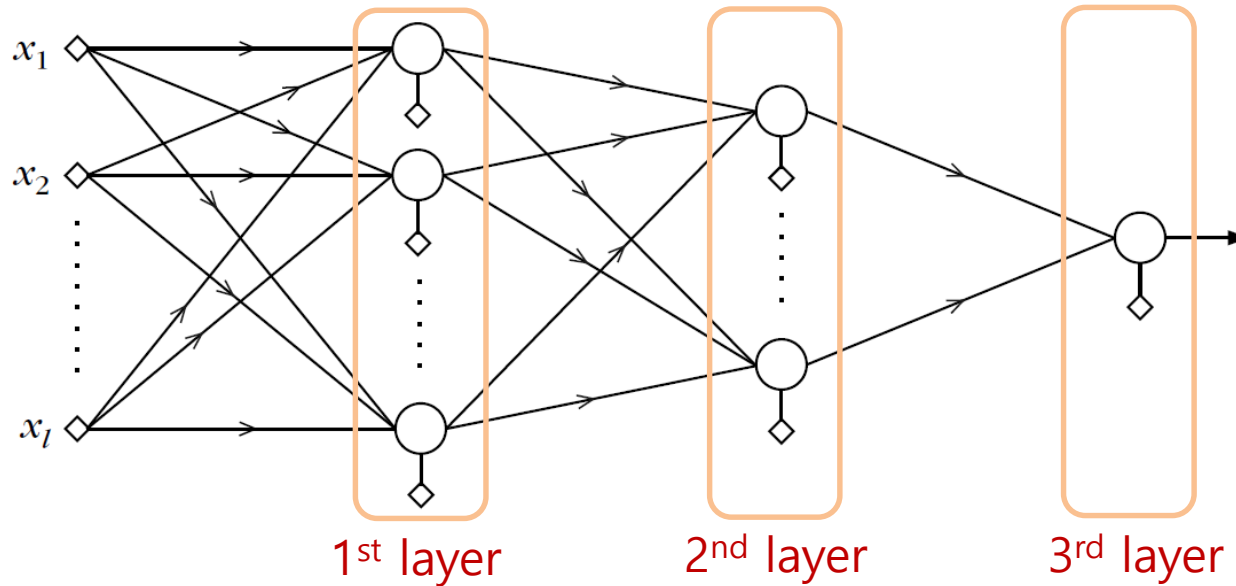
- Classification capabilities of three-layer perceptron

Three-layer perceptron can detect a class, which consists of **any** union of polyhedral regions



Three-Layer Perceptron

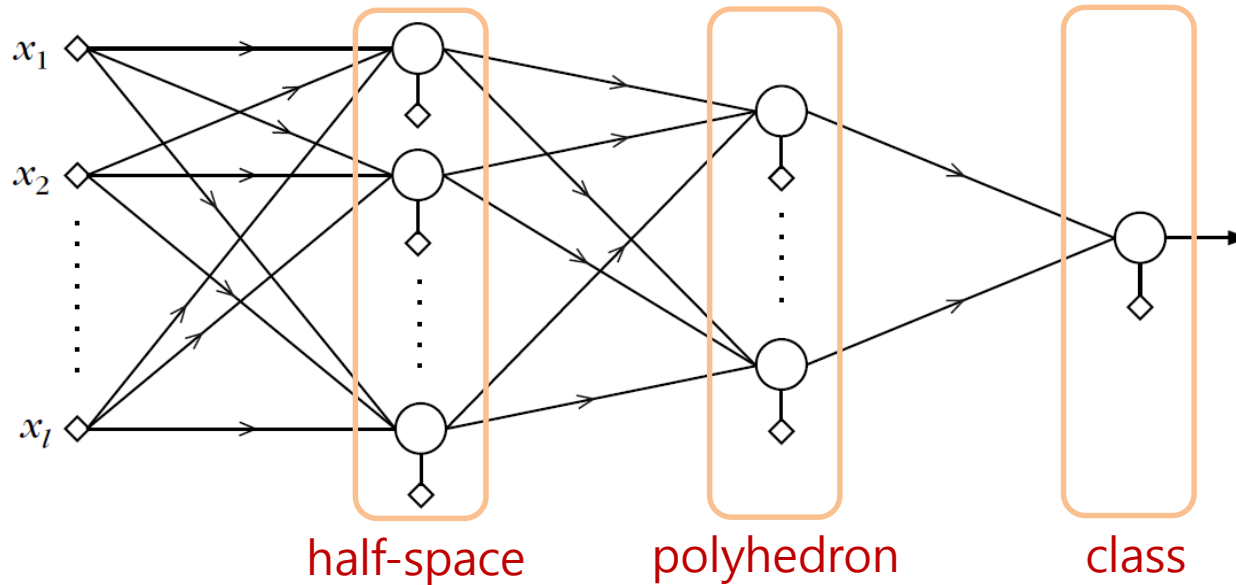
- Classification capabilities of three-layer perceptron



- In 2nd layer, for each neuron, the synaptic weights are chosen so that the realized hyperplane leaves only one of the H_p vertices on one side and all the rest on the other
- 3rd layer implements OR gate

Three-Layer Perceptron

- Classification capabilities of three-layer perceptron



- 1st layer detects half-spaces
- 2nd layer detects polyhedra
- 3rd layer detects a class, which is any union of polyhedra

BACKPROPAGATION ALGORITHM

Multilayer Perceptron Design

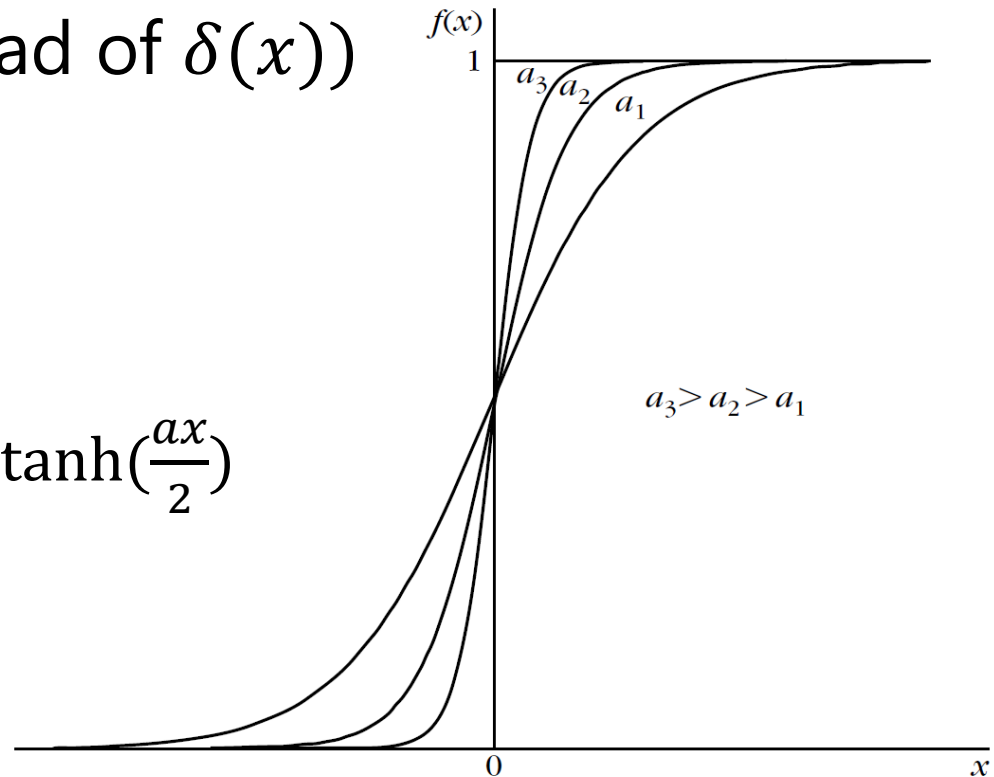
- Design a multilayer perceptron
 - Fix an architecture, and optimize the synaptic weights
 - To use the gradient descent scheme, we need a continuous activation function

- Logistic function (instead of $\delta(x)$)

- $f(x) = \frac{1}{1+\exp(-ax)}$

- $f(x) = \frac{2}{1+\exp(-ax)} - 1$

- $f(x) = c \frac{1-\exp(-ax)}{1+\exp(-ax)} = c \tanh\left(\frac{ax}{2}\right)$

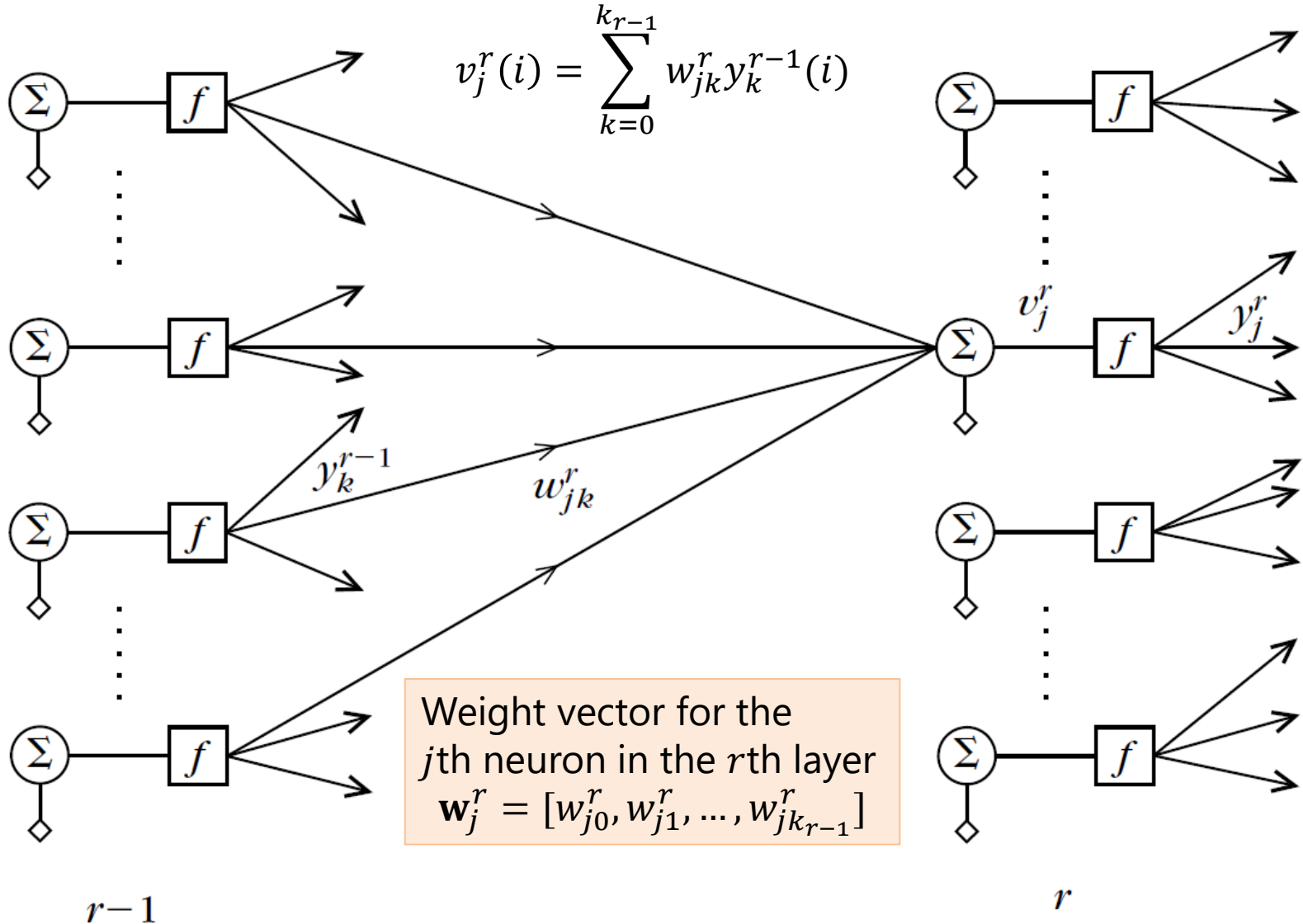


Architecture and Formulation

- L layers and k_r neurons in the r th layer ($r = 1, \dots, L$)
 - $k_0 = l$ nodes in the input layer
 - k_L output neurons
- N training pairs, $(\mathbf{y}(i), \mathbf{x}(i))$, $i = 1, \dots, N$, are available
 - $\mathbf{y}(i) = [y_1(i), \dots, y_{k_L}(i)]^T$
 - $\mathbf{x}(i) = [x_1(i), \dots, x_{k_0}(i)]^T$
- During training, the actual output $\hat{\mathbf{y}}(i)$ is different from the desired one $\mathbf{y}(i)$
- Compute the synaptic weights to minimize

$$J = \sum_{i=1}^N \mathcal{E}(i)$$
$$\mathcal{E}(i) = \frac{1}{2} \sum_{m=1}^{k_L} e_m^2(i) \equiv \frac{1}{2} \sum_{m=1}^{k_L} (\hat{y}_m(i) - y_m(i))^2$$

Definition of Variables



Gradient Descent

$$\mathbf{w}_j^r (\text{new}) = \mathbf{w}_j^r (\text{old}) + \Delta \mathbf{w}_j^r$$

$$\Delta \mathbf{w}_j^r = -\mu \frac{\partial J}{\partial \mathbf{w}_j^r} = -\mu \sum_{i=1}^N \delta_j^r(i) \mathbf{y}^{r-1}(i)$$

This is because

- $\frac{\partial \mathcal{E}(i)}{\partial \mathbf{w}_j^r} = \frac{\partial \mathcal{E}(i)}{\partial v_j^r(i)} \frac{\partial v_j^r(i)}{\partial \mathbf{w}_j^r} = \delta_j^r(i) \mathbf{y}^{r-1}(i)$
- $v_j^r(i) = \sum_{k=0}^{k_{r-1}} w_{jk}^r \mathbf{y}_k^{r-1}(i) = (\mathbf{w}_j^r)^T \mathbf{y}^{r-1}(i)$
- $\frac{\partial \mathcal{E}(i)}{\partial v_j^r(i)} \equiv \delta_j^r(i)$

Computing $\delta_j^r(i)$

1. $r = L$

$$\delta_j^L(i) = e_j(i) f' \left(v_j^L(i) \right)$$

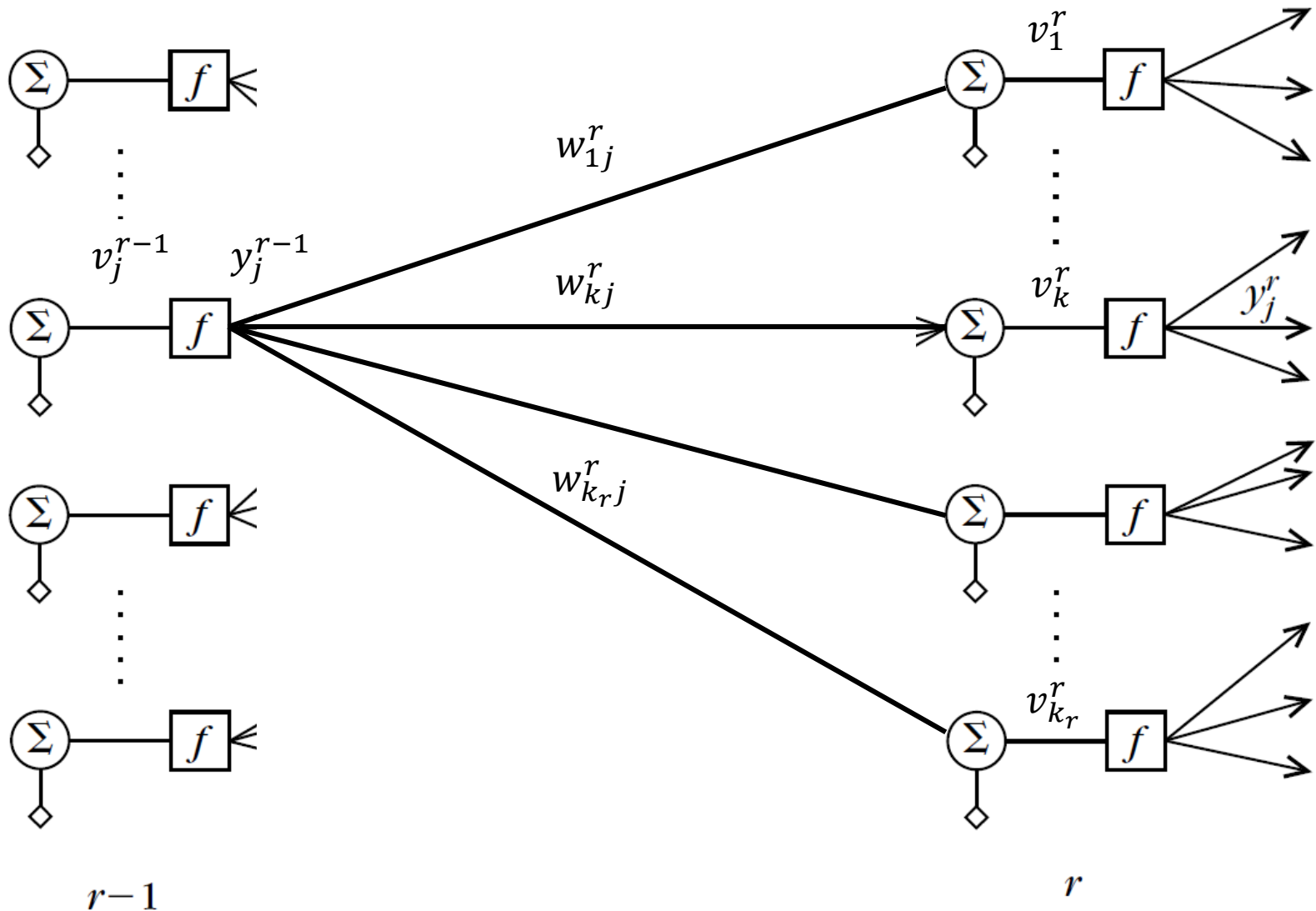
2. $r - 1 < L$

$$\delta_j^{r-1}(i) = e_j^{r-1}(i) f' \left(v_j^{r-1}(i) \right)$$

where

$$e_j^{r-1}(i) = \sum_{k=1}^{k_r} \delta_k^r(i) w_{kj}^r$$

$$\delta_j^{r-1}(i) = e_j^{r-1}(i) f' \left(v_j^{r-1}(i) \right) \quad \text{where} \quad e_j^{r-1}(i) = \sum_{k=1}^{k_r} \delta_k^r(i) w_{kj}^r$$



Backpropagation Algorithm

- **Initialization:** Initialize all weights with small random values.
- **Forward computations:** For each training vector $\mathbf{x}(i)$, compute all $v_j^r(i)$, $y_j^r(i) = f(v_j^r(i))$. Compute the cost function for the current estimate of weights.

- **Backward computations:**

- First,

$$\delta_j^L(i) = e_j(i) f'(v_j^L(i))$$

- Then, for $r = L, L - 1, \dots, 2$,

$$\delta_j^{r-1}(i) = e_j^{r-1}(i) f'(v_j^{r-1}(i)) \text{ where } e_j^{r-1}(i) = \sum_{k=1}^{k_r} \delta_k^r(i) w_{kj}^r$$

- **Weight update:**

$$\mathbf{w}_j^r(\text{new}) = \mathbf{w}_j^r(\text{old}) + \Delta \mathbf{w}_j^r$$

$$\Delta \mathbf{w}_j^r = -\mu \sum_{i=1}^N \delta_j^r(i) \mathbf{y}^{r-1}(i)$$

Remarks

- Termination
 - Cost function J becomes smaller than a threshold
 - Its gradients become small as compared with the weights
- Setting the learning constant μ
 - Tradeoff between fast convergence and overshooting
 - It depends on the shape of J
- Local minimum
 - Reinitialization

Remarks

- Batch mode vs pattern (online) mode

- Batch mode

$$\mathbf{w}_j^r(\text{new}) = \mathbf{w}_j^r(\text{old}) - \mu \sum_{i=1}^N \delta_j^r(i) \mathbf{y}^{r-1}(i)$$

- Well-behaved convergence due to the averaging

- Pattern mode

$$\mathbf{w}_j^r(i+1) = \mathbf{w}_j^r(i) - \mu \delta_j^r(i) \mathbf{y}^{r-1}(i)$$

- Randomness is helpful for avoiding local minima
 - White noise may be added to training data
 - In each epoch (one complete presentation of all N training pairs), the presentation order is randomized

- Testing is computationally more efficient than training

ISSUES IN MULTI-LAYER PERCEPTRON

Variations

- Momentum term

$$\Delta \mathbf{w}_j^r(\text{new}) = \alpha \Delta \mathbf{w}_j^r(\text{old}) - \mu \sum_{i=1}^N \delta_j^r(i) \mathbf{y}^{r-1}(i)$$
$$\mathbf{w}_j^r(\text{new}) = \mathbf{w}_j^r(\text{old}) + \Delta \mathbf{w}_j^r(\text{new})$$

- Momentum factor $\alpha \in [0.1, 0.8]$

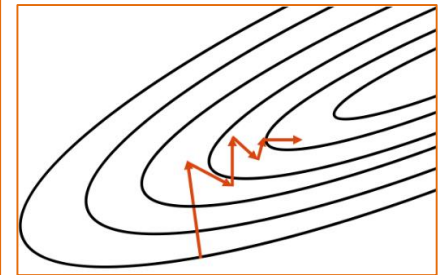
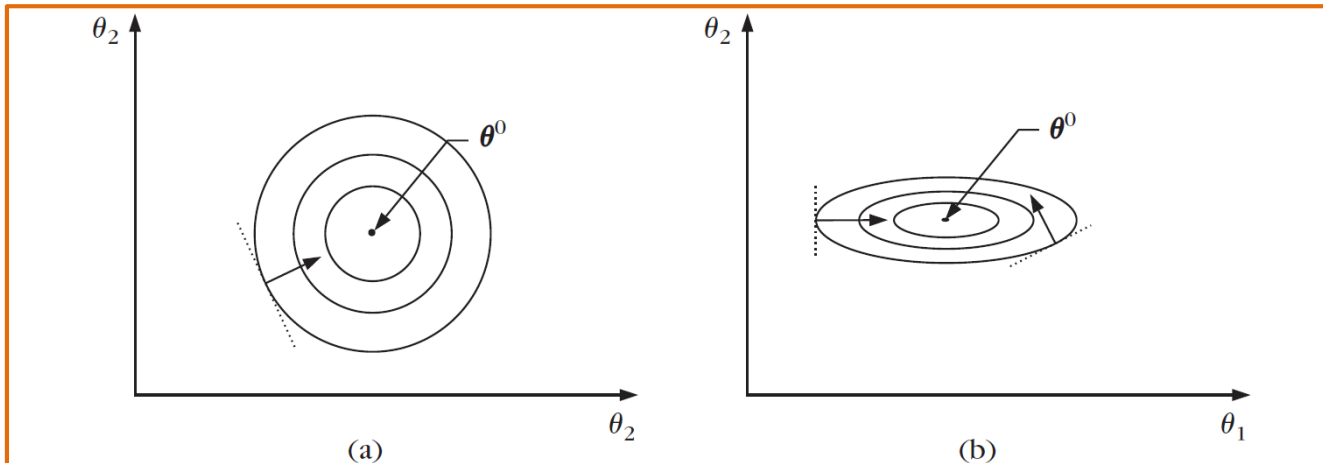


FIGURE C.3

Curves of constant cost values. In (a) the negative gradient always points to the optimum. In (b) it points to the optimum at only a few places, and convergence can be slow. The correction term can follow a zig zag path.

Variations

- Adaptive learning factor

$$\mu(t) = \begin{cases} 1.05\mu(t-1) & \text{if } \frac{J(t)}{J(t-1)} < 1 \\ 0.7\mu(t-1) & \text{if } \frac{J(t)}{J(t-1)} > 1.04 \\ \mu(t) & \text{otherwise} \end{cases}$$

Alternate Cost Functions

- Recall the least squares cost function

$$J = \frac{1}{2} \sum_{i=1}^N \sum_{k=1}^{k_L} (\hat{y}_k(i) - y_k(i))^2$$

- Assume

- y_k : desired class probability
- \hat{y}_k : obtained class probability with the softmax activation

$$\hat{y}_k = \frac{\exp(v_k^L)}{\sum_{k'} \exp(v_{k'}^L)}$$

- **Cross-entropy cost function**

$$J = - \sum_{i=1}^N \sum_{k=1}^{k_L} y_k(i) \ln \frac{\hat{y}_k(i)}{y_k(i)}$$

Alternate Cost Functions

- Cross-entropy cost function with softmax activation

$$J = - \sum_{i=1}^N \sum_{k=1}^{k_L} y_k(i) \ln \frac{\hat{y}_k(i)}{y_k(i)}$$

- This is minimized when $\hat{\mathbf{y}} = \mathbf{y}$
- If y_k is binary

$$J = - \sum_{i=1}^N \sum_{k=1}^{k_L} y_k(i) \ln \hat{y}_k(i)$$

- In the backpropagation,

$$\delta_j^L = \hat{y}_j - y_j$$

Network Size

- The smaller, the better
 - Generalization performance
 - When there are a large number of free parameters, the network tends to adapt to the particular details of the specific training set
 - Computational efficiency
 - For both training and testing
- Recent trend
 - Large network with big data

Network Size

- **Analytical method** for selecting the number of neurons
 - Ex) In the l -dimensional feature space, a single hidden layer with K neurons can form a maximum of M polyhedral regions,

$$M = \sum_{m=0}^l \binom{K}{m}$$

where $\binom{K}{m} = 0$ for $K < m$

Network Size

- **Pruning method I**

1. Train the network using the backpropagation algorithm for a number of iteration steps so that its cost function is reduced sufficiently
2. For the current weight estimates, compute the respective saliency values and remove those weights with small saliencies
3. Continue the training process with the remaining weights

- Computing the saliency s_i

$$s_i = \frac{b_{ii}w_i^2}{2}$$

$$- \delta J \cong \sum_i g_i \delta w_i + \frac{1}{2} \sum_i b_{ii} \delta w_i^2 + \frac{1}{2} \sum_{\substack{i,j \\ i \neq j}} b_{ij} \delta w_i \delta w_j \cong \frac{1}{2} \sum_i b_{ii} \delta w_i^2$$

$$\text{where } g_i = \frac{\partial J}{\partial w_i}, b_{ij} = \frac{\partial^2 J}{\partial w_i \partial w_j} \text{ (backpropagation)}$$

Network Size

- **Pruning method II**

1. Train the network using the backpropagation algorithm for a number of iteration steps so that its cost function is reduced sufficiently

$$J = \sum_{i=1}^N \mathcal{E}(i) + \alpha \mathcal{E}_p(\mathbf{w})$$

- Regularization term $\mathcal{E}_p(\mathbf{w}) = \sum_{k=1}^K h(w_k^2)$ where $h(w^2) = \frac{w^2}{w_0^2 + w^2}$
2. Prune the weights that are smaller than a pre-specified threshold
 3. Continue the training process with the remaining weights

Overtraining

- Similar to overfitting

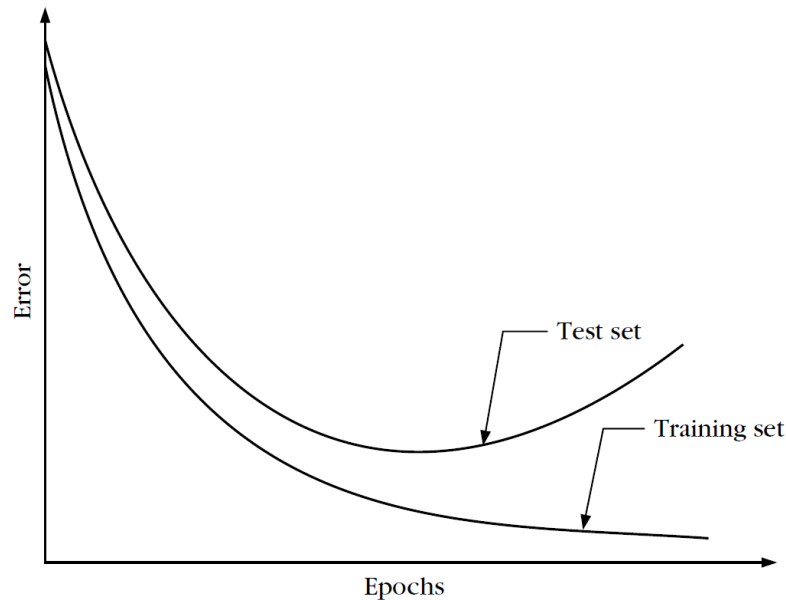


FIGURE 4.14

Trend of the output error versus the number of epochs illustrating overtraining of the training set.

- To avoid it, divide an available dataset into training and validation sets and validate the system

Simulation Example

- 3-layer perceptron (3, 2, 1 neurons in 1st, 2nd, 3rd layers)
 - $\mu = 0.05$, $\alpha = 0.85$
 - The momentum with adaptive μ is faster than the ordinary momentum

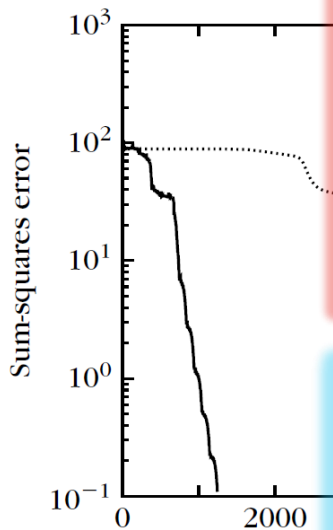


FIGURE 4.15

(a) Error convergence curves for two algorithms. Note that the adaptive momentum algorithm converges faster than the ordinary momentum algorithm. The network was formed by the multilayer perceptron.

- Momentum term

$$\Delta \mathbf{w}_j^r(\text{new}) = \alpha \Delta \mathbf{w}_j^r(\text{old}) - \mu \sum_{i=1}^N \delta_j^r(i) \mathbf{y}^{r-1}(i)$$

$$\mathbf{w}_j^r(\text{new}) = \mathbf{w}_j^r(\text{old}) + \Delta \mathbf{w}_j^r(\text{new})$$

- Momentum factor $\alpha \in [0.1, 0.8]$

- Adaptive learning factor

$$\mu(t) = \begin{cases} 1.05\mu(t-1) & \text{if } \frac{J(t)}{J(t-1)} < 1 \\ 0.7\mu(t-1) & \text{if } \frac{J(t)}{J(t-1)} > 1.04 \\ \mu(t) & \text{otherwise} \end{cases}$$

Simulation Example

- 3-layer perceptron (3, 2, 1 neurons in 1st, 2nd, 3rd layers)
 - $\mu = 0.05$, $\alpha = 0.85$
 - The momentum with adaptive μ is faster than the ordinary momentum

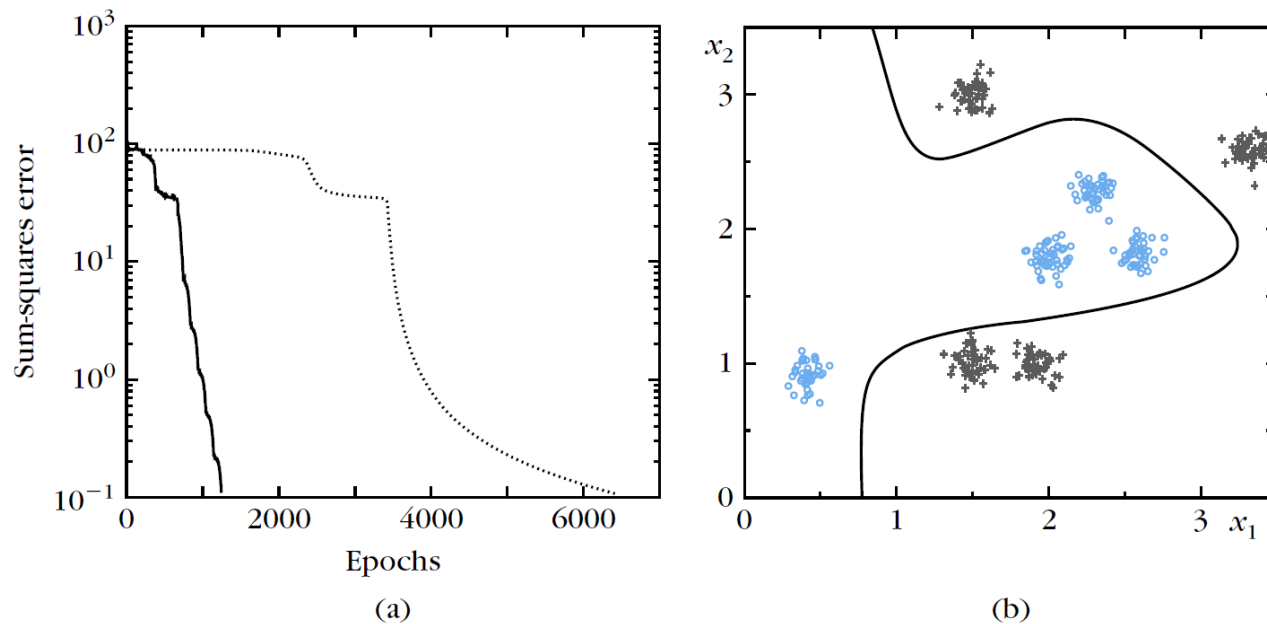


FIGURE 4.15

(a) Error convergence curves for the adaptive momentum (dark line) and the momentum algorithms. Note that the adaptive momentum leads to faster convergence. (b) The decision curve formed by the multilayer perceptron.

Simulation Example

- 3-layer perceptron (20, 20, 1 neurons in 1st, 2nd, 3rd layers)
 - Saliency-based pruning: 480 parameters to 25 parameters

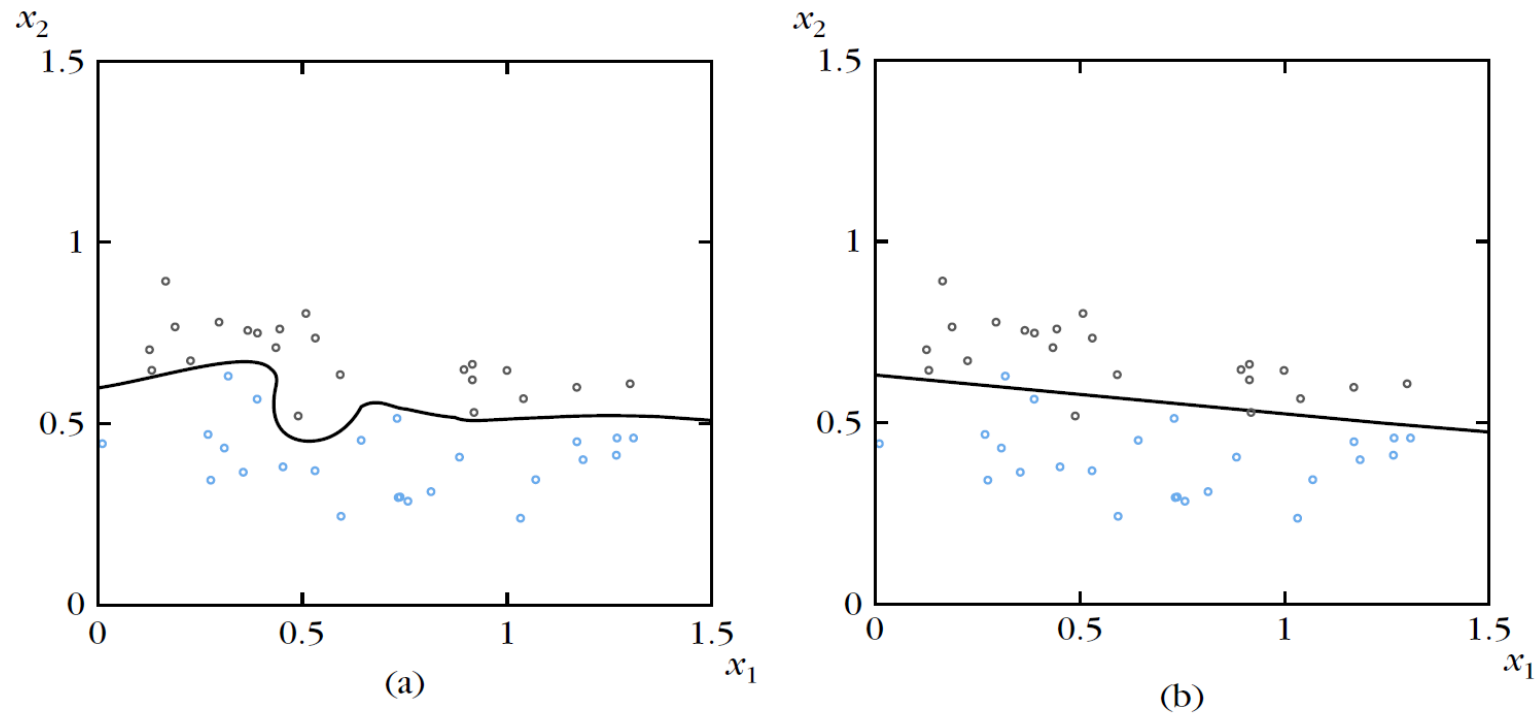
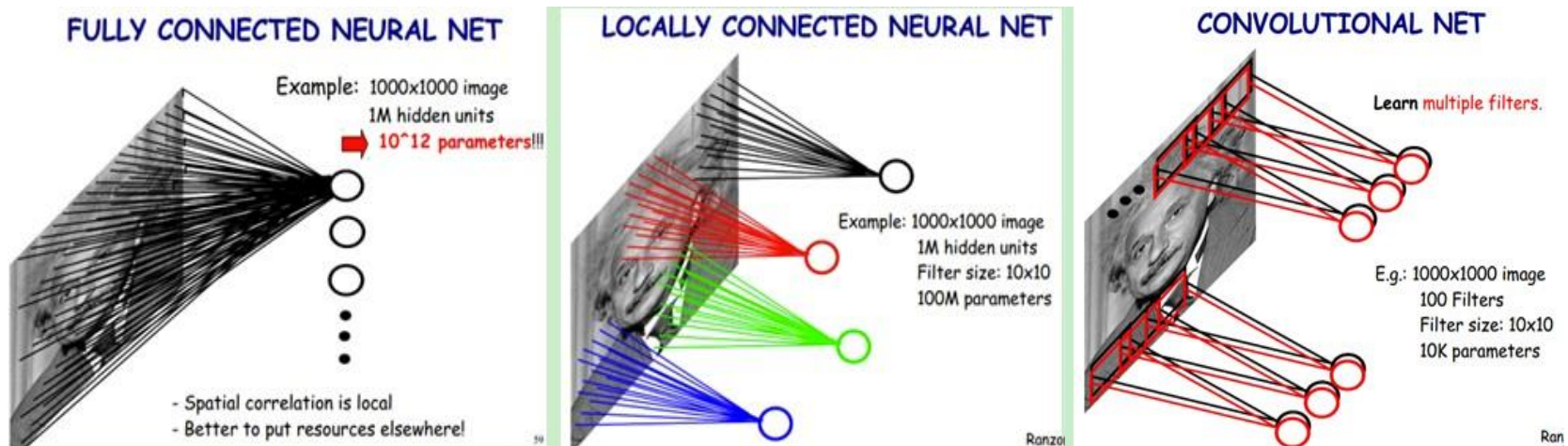


FIGURE 4.16

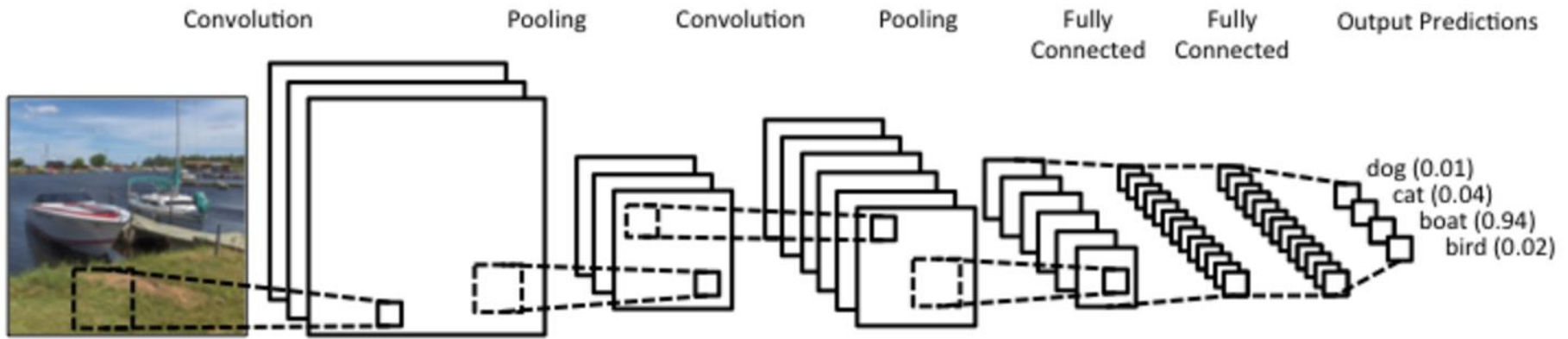
Decision curve (a) before pruning and (b) after pruning.

Network with Weight Sharing

- Ex) Convolutional neural network (CNN)



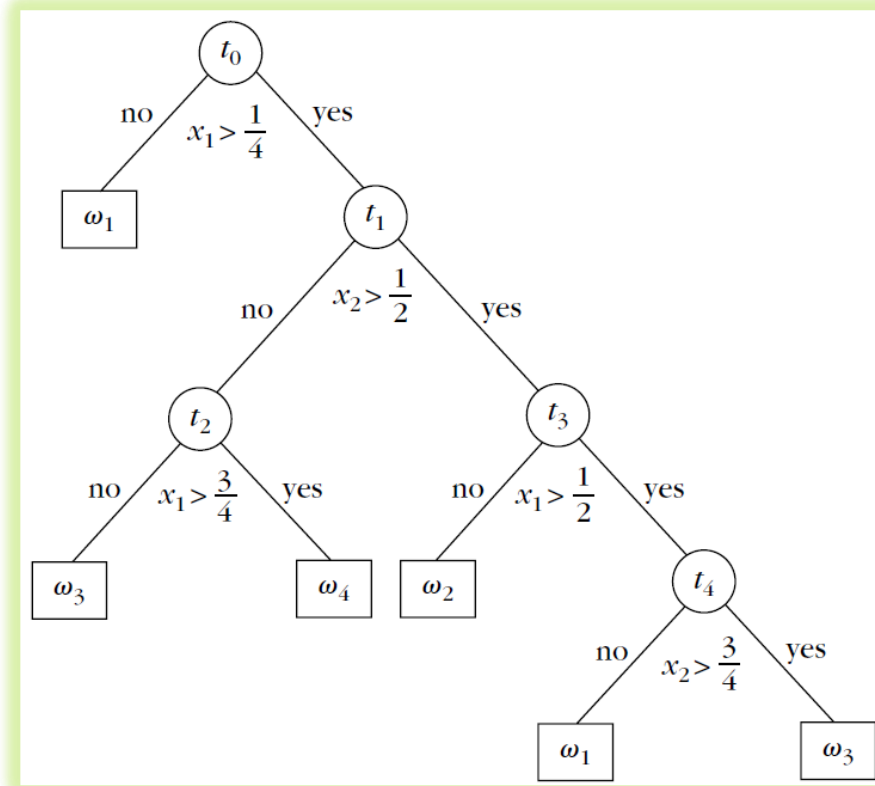
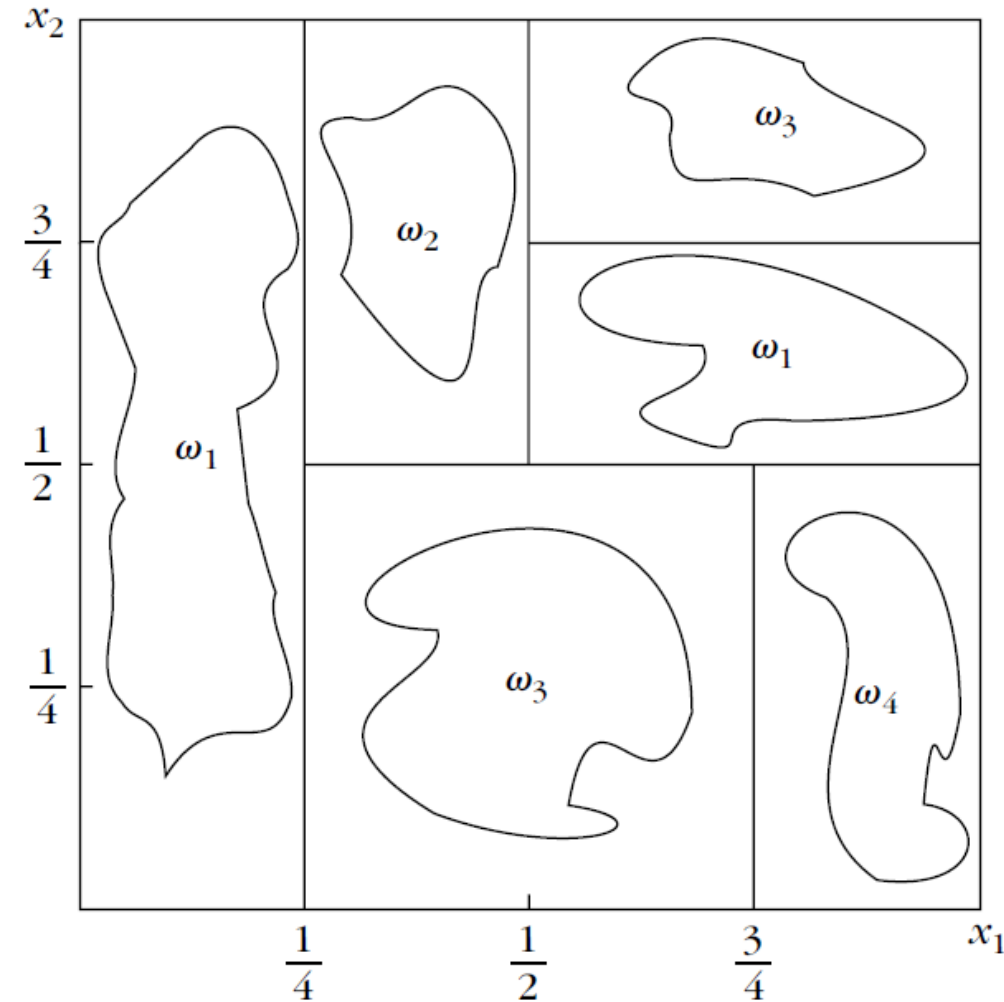
Convolutional Neural Network



DECISION TREES

Multistage Decision System

- A Sequence of Decisions



OBCT: Ordinary Binary Classification Tree
 Each decision splits the space into two hyper-rectangles

OBCT Training

- Set of questions
 - Each node t is associated with a subset X_t of the training set X
 - It is partitioned into X_{tY} and X_{tN} by a question
 - $X_{tY} \cap X_{tN} = \emptyset$ and $X_{tY} \cup X_{tN} = X_t$
 - What is the set of possible questions?
- Splitting criterion
- Stop-splitting rule
- Class assignment rule

OBCT Training

- **Set of questions**

- Note that this set is finite

- **Splitting criterion**

- Impurity of node t

$$I(t) = - \sum_{i=1}^M P(\omega_i|t) \log P(\omega_i|t)$$

- Impurity decrease after the splitting

$$\Delta I(t) = I(t) - \frac{N_{tY}}{N_t} I(t_Y) - \frac{N_{tN}}{N_t} I(t_N)$$

- Select the question with the maximum $\Delta I(t)$

Example 4.2

In a tree classification task, the set X_t , associated with node t , contains $N_t = 10$ vectors. Four of these belong to class ω_1 , four to class ω_2 , and two to class ω_3 , in a three-class classification task. The node splitting results into two new subsets X_{tY} , with three vectors from ω_1 , and one from ω_2 , and X_{tN} with one vector from ω_1 , three from ω_2 , and two from ω_3 . The goal is to compute the decrease in node impurity after splitting.

We have that

$$I(t) = -\frac{4}{10} \log_2 \frac{4}{10} - \frac{4}{10} \log_2 \frac{4}{10} - \frac{2}{10} \log_2 \frac{2}{10} = 1.521$$

$$I(t_Y) = -\frac{3}{4} \log_2 \frac{3}{4} - \frac{1}{4} \log_2 \frac{1}{4} = 0.815$$

$$I(t_N) = -\frac{1}{6} \log_2 \frac{1}{6} - \frac{3}{6} \log_2 \frac{3}{6} - \frac{2}{6} \log_2 \frac{2}{6} = 1.472$$

Hence, the impurity decrease after splitting is

$$\Delta I(t) = 1.521 - \frac{4}{10}(0.815) - \frac{6}{10}(1.472) = 0.315$$

OBCT Training

- **Stop-splitting rule:** Stop if
 - the maximum $\Delta I(t)$ is less than a threshold, or
 - $|X_t|$ is small enough, or
 - X_t is pure
- **Class assignment rule**
$$j = \arg \max_i P(\omega_i | t)$$

OBCT Training

- Begin with the root node $X_t = X$
- **For** each new node t
 - **For** every feature $x_k, k = 1, \dots, l$
 - **For** every value $\alpha_n, n = 1, \dots, N_{tk}$
 - Generate X_{tY} and X_{tN} according to $x_k(i) \leq \alpha_n, i = 1, \dots, N_t$
 - Compute the impurity decrease
 - **End**
 - Choose α_n with the maximum impurity decrease
 - **End**
 - Choose x_{k^*} and α^*
 - **If** the stop-splitting rule is met
 - Declare t as a leaf and designate it with a class label
 - **Else**
 - Generate nodes t_Y and t_N according to $x_{k^*}(i) \leq \alpha^*$
- **End**

OBCT: Remarks

- Tree size must be large enough but not too large
 - Grow a tree up to a large size first and then prune it
- Improving tree classifiers
 - High variance issue: a small change in the training data set result in a very different tree
 - Bagging (bootstrap aggregating)
 - Create variants X_1, \dots, X_B of the training set X , by sampling with replacement
 - For each variant, design a tree
 - The final decision is obtained by the majority rule using votes from the B decision trees
 - **Random forests** use the idea of bagging with random feature selection

COMBINING CLASSIFIERS

Given $P_j(\omega_i|\mathbf{x})$, find an improved estimate $P(\omega_i|\mathbf{x})$

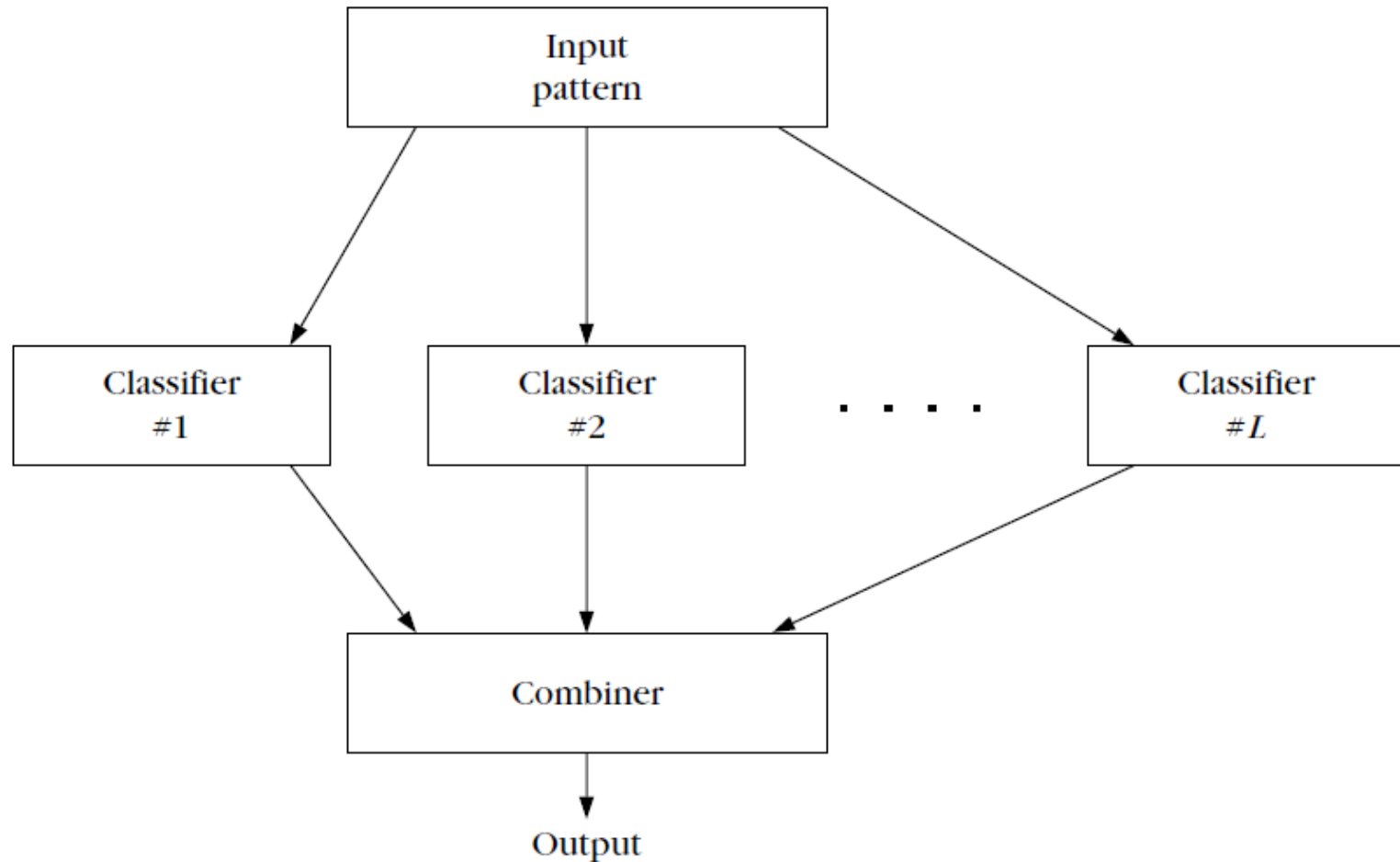


FIGURE 4.29

L classifiers are combined in order to provide the final decision for an input pattern. The individual classifiers may operate in the same or in different feature spaces.

Geometric Average Rule

$$P(\omega_i|\mathbf{x}) = C \prod_{j=1}^L (P_j(\omega_i|x))^{\frac{1}{L}}$$

- Note that this minimizes

$$D = \frac{1}{L} \sum_{j=1}^L \sum_{i=1}^M P(\omega_i|\mathbf{x}) \log \frac{P(\omega_i|\mathbf{x})}{P_j(\omega_i|\mathbf{x})}$$

Arithmetic Average Rule

$$P(\omega_i|\mathbf{x}) = \frac{1}{L} \sum_{j=1}^L P_j(\omega_i|\mathbf{x})$$

- Note that this minimizes

$$D = \frac{1}{L} \sum_{j=1}^L \sum_{i=1}^M P_j(\omega_i|\mathbf{x}) \log \frac{P_j(\omega_i|\mathbf{x})}{P(\omega_i|\mathbf{x})}$$

Majority Voting Rule

- Decides in favor of the class, when at least l_c classifiers agree on the class label

$$l_c = \begin{cases} \frac{L}{2} + 1 & L \text{ even} \\ \frac{L+1}{2} & L \text{ odd} \end{cases}$$

- Assumption
 - The number of classifiers L is odd
 - Each classifier has the same probability p of correct classification
 - The decision of each classifier is independent

- Some properties

$$P_c(L) = \sum_{m=l_c}^L \binom{L}{m} p^m (1-p)^{L-m}$$

- If $p > 0.5$, $P_c(L)$ is monotonically increasing in L and $P_c(L) \rightarrow 1$ as $L \rightarrow \infty$
- If $p < 0.5$, $P_c(L)$ is monotonically decreasing in L and $P_c(L) \rightarrow 0$ as $L \rightarrow \infty$
- If $p = 0.5$, $P_c(L) = 0.5$ for all L